Context-Dependent Joint-Decision Arbitration for Computer Games

Gal A. Kaminka, Jared Go and Thuc D. Vu

Computer Science Department
Carnegie Mellon University
galk@cs.cmu.edu, {jgo,tdv}@andrew.cmu.edu

Abstract. Multi-agent teams benefit from the application of explicit teamwork mechanisms. These alleviate the workload on the designer by automating key components of teamwork, such as communication content and timing decisions, conflict resolution, and joint decision making. However, existing mechanisms make architectural commitments to *context-free* methods of joint decision making: The agents use the same procedure (e.g., negotiations, voting, reference to rank) to decide on the next step to take, regardless of the status of their task, or the options available. We hypothesize that teams will benefit instead from *context-dependent arbitration*, where the selection of a joint-decision procedure depends on the status of the task and agents involved. To examine this hypothesis, this paper presents SCORE (Synchronous CoORdination Engine), a prototype teamwork and coordination executable model, that allows the human designer of a team to specify different joint-decision procedures and the conditions under which they are to be used. We evaluate SCORE and context-dependent arbitration in a complex multi-agent 3D virtual environment, using the GameBots interface. We empirically show that, all else being equal, using context-dependent arbitration results in performance which is superior to that resulting from context-free arbitration scheme.

1 Introduction

There is growing recognition, both in theory and in practice, that multi-agent teams can significantly benefit from the application of explicit teamwork mechanisms [2,5,4,13]. Such mechanisms can automate communication content and timing decisions, negotiations over joint decision making, coordination of activities, social organization into roles, and re-organization upon failures. This allows the human designer to focus on specifying the goal-oriented behavior of the agents, as collaborative behavior is automated to a large degree. Indeed, a number of teamwork models have been deployed successfully in complex dynamic multi-agent applications, e.g. GRATE* [5], COLLAGEN [11,9], STEAM steam-jair, and ALLIANCE [10].

A key benefit of teamwork models is their capacity for automating the team's decision-making and conflict resolution. The models provide communication and negotiation protocols that automate the team's arrival at an agreement as to what action-step, plan, or behavior is to be jointly executed by team-members. This frees the designer from the need to specify the communication procedures to be used in all possible circumstances. For instance, STEAM [13] allows any agent to announce the achievement or unachievability of a joint-subgoal, thus allowing any team-member to cause an abandonment of the relevant goal-oriented activities of its teammates. However, selection of the next subgoal to be tackled is left to the team-leader, who is responsible for beginning a confirm-request protocol to gain mutual belief in this new subgoal. Any conflicts (e.g., about which subgoal is to be pursued next) are handled in STEAM by referring to rank, though some decisions are implicitly resolved by referring to designer-determined roles (i.e., agents of two different roles select different plans).

Formally, an automated decision-making procedure (e.g., reference to rank, role-based, or negotiations) can be though of as a function that maps world and agents' states into a joint decision. There are many different realizations for such functions [15]: Auctions, reference to rank or to a central authority, argumentation [8,14], voting, market-based methods, etc.

However, while a plethora of procedures exists, a close examination of existing teamwork models reveals architectural commitments to context-free methods of arbitration, i.e., to a single procedure that ignores execution context. We introduce the term *arbitration policy* to denote a function which maps world and agents' states into a decision about which arbitration procedure is to be used. Current teamwork models are characterized by context-free arbitration policies: The same joint-decision procedure is used regardless of world and agents' states.

We hypothesize, however, that teamwork benefits from flexibility in the way that agents resolve conflicts and make joint decisions: Different mechanisms are appropriate depending on the context of task execution. For instance, *reference to rank*–a mechanism by which the agent with higher rank makes the decision centrally–may be appropriate if the agent with higher rank has access to better information. However, a voting scheme may be more appropriate when there is little time pressure, and all agents have access to the same information.

This paper examines *context-dependent* arbitration policies, in which a different procedure is used by the agents depending on the context of task execution, e.g., the agents may used role-based selection for one decision, and bidding in others. While ideally the arbitration policy itself would be automatically generated (leaving the decision about which procedures should be used to the teamwork model), for the purposes of testing our hypothesis we use manually-designed policies, in which the selection of a joint-decision protocol (e.g., negotiations, voting, etc.) is left to the designer.

To provide concrete empiric evaluation of context-dependent arbitration policies, we present a team of software agents in a complex, 3D multi-agent virtual environment, built using the GameBots interface [7]. The team is executing a dynamic team task, playing capture-the-flag against a different fixed team. To control their coordinated execution our agents use SCORE (Synchronous COoRdination Engine), a teamwork model that facilitates and manages coordinated teamwork, and allows the use of context-dependent arbitration policies.

We compare context-free and context-dependent arbitration policies and show that the context-dependent arbitration policy, which uses different conflict-resolution procedures depending on the context, is superior to the context-free arbitration policies using either one of the procedures, on a number of performance measure. Furthermore, the context-dependent arbitration policy results in less wasteful behavior by the team, so that each concentrated effort at the task is more productive.

This paper is organized as follows: Section 2 presents motivation and related work. Section 3 discusses SCORE, the run-time engine that enables context-dependent joint-behavior arbitration. Section 4 presents the experimental set-up and the results. Section 5 presents related work. Section 6 concludes.

2 Motivation and Background

The motivation for this work comes from our experience in constructing teams of autonomous software agents which play capture-the-flag (CTF) in a multi-agent 3D virtual environment, using the GameBots interface [7]. In this highly dynamic game, two teams of agents are each attempting to steal the opponent's flag from the opponent's base, and bring the flag back to the team's own home base. Each successful capture adds a point to the team's total. Agents have to navigate a complex environment, avoiding obstacles, and coordinating with each other. For instance, they may have to decide to leave some agents behind (to defend the home base) while others go forward to try to capture the opponent's flag. Or, the agents may decide to coordinate their capture attempt so that one agent clears the way for the other agent to go forward and steal the flag. Agents can communicate with their teammates and opponents, and use simulated magical wands to *tag* each other: A tag causes an agent to disappear from its current location and re-appear in one of a number of possible locations back at the home-base.

The task of constructing teams of agents for this task is quite challenging. A physical simulation server manages the simulation of all 3D physical objects, including agent bodies, their magical wands, and their surroundings. Each agent is a separate program, which connects to the server using a socket interface, through which the agent program receives sensory information, and sends back actuation information. Agents can see objects within a limited field of view, and can also hear sounds of movement or wand tagging. Agents can turn around, run or walk, and tag opponents and friends with the magical wands. They may also pick up health packs and wand replacements, which are sometimes found in the environment. In other words, each agent program provides the "brains" for a simulated "body". Since the body's sensing and action is limited, and since the environment is complex, the number of possible states is enormous.

To address the challenges of constructing agents for this domain, we chose to rely on two proven technologies: (i) behavior-based control for the design of the single agent, e.g., [3], and (ii) explicit teamwork models for the automated coordination of the agents, e.g., [13]. Using these two technologies, the designer first builds a behavior-hierarchy, specifying the steps to be taken by an agent in playing CTF (see next section for our chosen hierarchical behavior representation). Then, the designer specifies roles in the team, by constructing an organizational hierarchy separating subteams and allocating different agents to different subteams. Finally, the designer associates roles and subteams with specific behaviors. This establishes two classes of behaviors: *Team behaviors* are to be executed by their associated subteams together—all subteam members selecting and deselecting the behaviors at the same time (e.g., for coordinated attacks); *individual behaviors*, typically in service of team behaviors, can be selected by individual agents independently from the selections of their teammates. Thanks to the run-time support of a teamwork model such as STEAM [13], all communications required to synchronize the execution of team behaviors are automated. The designer does not need to write countless special-purpose rules that cover the many possible scenarios. Instead, the designer only specifies what behaviors are to be executed jointly, and the run-time teamwork model takes care of the joint-decision making by the agents.

Our initial implementation of the teamwork model for the agents built on the STEAM model [13]. Any agent may cause termination of a joint behavior, however when agents need to select a new team behavior for joint execution, a pre-defined team-leader makes a centralized decision on the next behavior to be selected, and then requests confirmation from its teammates[1]. Once all agents confirm, the team begins joint execution of the newly selected joint behavior. Thus in principle all conflicts are resolved by reference to the rank of the leader, i.e., all joint behavior arbitration is always done using the same method (in particular, request-confirm).

Leaving all decisions to a team-leader presented mixed results. On one hand, the decision making procedure was *quick*, for instance in assigning tasks to agents (e.g., escorting the carrier after stealing the flag vs. distracting opponents). Such soft real-time responsiveness was very appropriate given the dynamic competitive nature of the game. On the other hand, due to its own sensory limitations, the leader would sometimes make obviously poor decisions, such as assigning the task of clearing a path to the opponent base to an agent that was far away from it, while assigning the task of stealing the flag to an agent that was much closer—resulting in the path clearing being too late to be effective. A different decision-making procedure, e.g. one that would give more considerations to the agents' own preferences may alleviate these problems. On the other, such a procedure will cause a greater delay, as agents will have to discuss their preferences until they settle on a joint decision.

Taking our inspiration from human social organizations, we hypothesized that teams of agents will benefit from utilizing different decision-making proce-

[1] We simplified the model by leaving out STEAM's decision-theoretic evaluation of the costs of communications versus the costs of coordination. Thus our model triggers communications even when STEAM's may not have.

dures depending on the context for their decision. Just as a team of programmers may sometimes engage in lengthy discussions on the best course of action, and sometimes yield to the authority of a project leader, we believe that a team of agents can benefit from a context-dependent arbitration policy.

3 The Synchronous Coordination Engine

SCORE (Synchronous CoORdination Engine) is a teamwork and coordination engine for multi-agent teams using behavior-based control. SCORE handles the tasks of team synchronization, and context-dependent joint-behavior arbitration under designer-specified task constraints. To do this, SCORE automates and manages communications between agents, controlling content and timing to control coordinated execution. This section opens with a brief overview of the behavior-based control methodology used in building the CTF team, including a few short illustrative examples (Section 3.1). Then, we describe SCORE's team synchronization (Section 3.2) and context-dependent arbitration mechanisms (Section 3.3). Finally, Section 3.4 provides an overview of the team behavior specification language, parsed by SCORE, which ties the different mechanisms together.

3.1 Single-Agent Behavior-Based Control

To facilitate responsive execution in the dynamic environment in which the CTF-playing agents are to participate, we chose to implement a behavior-based control system. This system is similar in many ways to others that have been reported in the literature (e.g., RAPs [3]) and so will only be described here briefly.

The control of each individual agent is divided into a set of behaviors, each one responsible for its own sensing requirements, memory/state maintenance, and action scheduling. Each behavior is associated with selection conditions and termination conditions, typically testing environmental features but also possibly testing globally-set variables. When selection conditions are satisfied, the behavior is selected for execution, and it takes complete control of the agent involved. When its termination conditions are satisfied, it removes itself from control and allows other behaviors to be selected. Conflicts in selection are handled by designer-specified control rules.

The designer, on top of selection and termination conditions, can provide temporal execution constraints, which impose a (partial) order on the selection of behaviors, and specifies which behaviors are to follow others. Such temporal execution constraints can be cyclic, allowing a behavior to follow itself or its own predecessors in order of execution). Behaviors can also be hierarchically decomposed into other behaviors, in order to facilitate meaningful design compositions (such as goal/subgoal), and primitive behavior re-use in service of different super-behaviors. Such hierarchical decomposition links point from a behavior to its *first* child(ren), i.e., any children behaviors which begin a temporally-constrained execution. In our system, hierarchical links may not be cyclic. Multiple temporal or hierarchical outgoing links denote alternative paths of execution.

For instance, Figure 1 presents the behavior hierarchy for the CTF team. Here, for instance, the `Explore` behavior is to be followed by the `WinGame` behavior. This last behavior has two alternative decompositions, into the `Defend` behavior sub-tree, and into the `Attack` behavior sub-tree. Agents choosing `Defend` will choose either `Return-our-flag` or `Defend-Base`. When these are terminated, the agents will agent choose `Defend` and again will have to choose between its children as appropriate. On the other hand, Agents choosing the `Attack` subtree will start with the `Attack` behavior (and either one of its children), and will then follow it with a `Go-back-to-our-base` behavior. Upon its termination, the agents will choose `Attack` again.

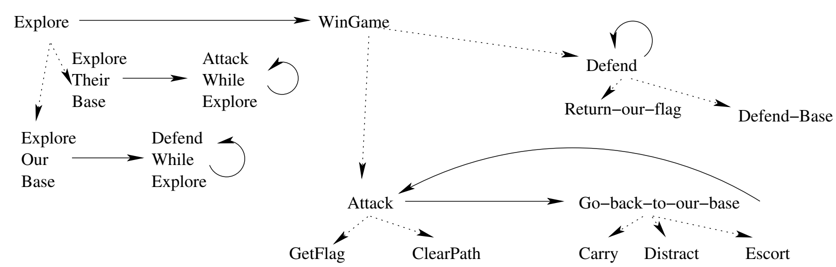

Fig. 1. Behavior Hierarchy for the CTF team. Full arrow lines mark temporal order constraints. Dotted arrow lines mark hierarchical decomposition links.

3.2 Maintaining Team Synchronization

The single-agent behavior-based control described above is insufficient by itself to manage highly coordinated *joint* behaviors, i.e., behaviors that are to be executed at the same time by several teammates. Some of the behaviors, e.g., `Attack`, may be intended for joint execution, and so appropriately tagged by the designer as *team behaviors.* For instance, all relevant agents may jointly select `Attack`, where by some members select the `Go-to-flag` behavior while others select the `Attack-Base` behavior (see Figure 1). However, this relies on the relevant agents to each select and de-select their own instances of the appropriate behaviors at approximately the same time.

Before the introduction of explicit teamwork models, all communications related to such decisions had to be specifically provided by the designer. However, in dynamic domains the number of possible different states that an agent may find itself with respect to its teammates prohibits attempts at fully enumerating all appropriate responses a-priori [13]. Explicit teamwork models were introduced to automatically manage communications among agents so as to alleviate the need to foresee all possible inter-agent situations. Teamwork theories, such as *Joint Intentions* [2] and *SharedPlans* [4] have provided principles of teamwork which could be translated into rules governing communications to manage agreement and conflict resolution.

SCORE builds on previous work on explicit teamwork models to ensure that when an individual member of a (sub)team selects a behavior for its own execution, and that behavior is tagged by the designer as a team-behavior (i.e., to be executed in agreement with the other members of the team), then the other agents select the agreed-upon behavior at the same time. For instance, suppose a team-member privately believes that the `Explore` behavior is to be terminated (since its termination conditions were satisfied). Since `Explore` is tagged by the designer as a team-behavior, ideally all relevant members of the team should be terminating it at the same time.

SCORE provides a synchronization service which is automatically invoked. The service maintains a table of *team variables*, whose value SCORE seeks to replicate across the different agents by communicating when the values change. For instance, the `Explore` behavior is terminated when two location variables are known: The location of the team's own base, and the location of the opponent base. The two variables representing these locations are put in the team-synchronized variable table: From that point on, any change in their values (by one of the agent) will automatically be propagated by SCORE to the other agents. Thus once an agent discover the locations of the two bases, it will terminate its own execution of the `Explore` behavior. However, due to the replication of the location variables, the position will be propagated automatically to the other team-members, which will cause them to terminate their own execution of the `Explore` behavior as well (since their own termination conditions will have been satisfied).

This method may face difficulties as the number of agents is scaled up, and it relies on the assumption that agents will not have conflicting beliefs in the values for team-synchronized variables (e.g., that two different agents discover the base location in two different places). However, we find that for the purposes of computer games, specifically involving small teams and reliable communications, it is simple and efficient. Indeed, similar methods have been successfully used in the past in domains with similar characteristics, such as RoboCup [1,12].

3.3 Context Dependent Arbitration

The team-variable synchronization described above can be useful in both jointly terminating and selecting behaviors by relevant agents. However, it does not at all deal with conflict resolution and joint behavior arbitration. For instance, suppose all agents select `WinGame` for execution. While all should execute `WinGame`, only some agents should execute `Attack`, while others should execute `Defend`. Naturally, different agents may have different beliefs as to whom should execute what behavior. This is where arbitration is to take place.

SCORE differentiates itself from other teamwork models by allowing for context-dependent arbitration. While models such as STEAM utilize a fixed procedure by which agents come to decide on their selected behaviors, SCORE allows the designer to specify a different procedure to be used, depending on the execution context. Thus depending on the state of the world and agents involved, a different arbitration procedure may be used. As with joint behaviors, each agent runs its own copy of the specified arbitrator, which manages communications with the other agents' arbitrators to carry out the arbitration procedure. For now, we assume the arbitrator chosen by the designer is known by all agents, so that they all use the same arbitration procedure (much like we assume they all have access to the same joint behaviors).

To allow for such flexibility in arbitration procedures, SCORE internally defines an *arbitration API*, which allows new joint-behavior arbitration procedures (henceforth, *arbitrators*) to be plugged into SCORE seamlessly: The inputs to arbitrators include the state of the world (including currently executing behaviors and beliefs about the agent and its teammates), and the alternative behaviors of which one is to be selected. The output expected is a choice of one of the alternative behaviors.

As a first step to utilizing different multiple arbitrators, SCORE allows the designer to specify which arbitrator to use in initiating which behavior, thus making the arbitrator selection dependent on the task execution context. We have defined two significantly different arbitrators, which we will use in different combinations to evaluate this approach (Section 4). The designer has the choice of requiring the use of either one of these arbitrators. In context-free arbitration, only one arbitrator would be available.

The first arbitrator is based on designer-specified roles: The assumption here is that the designer assigns names roles to each agent, and that these roles are then associated with specific decisions with respect to joint behaviors. For instance, one agent in the team may be assigned the role of Attacker, and will always select specific behaviors, while another agent may be assigned the role of a Defender, and will always select other behaviors.

To carry out role-based arbitration, SCORE relies on a designer-specified organization hierarchy, which specifies team/subteam relationships, and places individual agents within specific subteams, thus defining their roles. For instance, Figure 2 presents the organization hierarchy for the our CTF team. Three subteams are created by the designer: Defenders, Attackers, and SecondAttackers. The designer may choose the number of agents available in each subteam. The designer must also specify which behaviors in the behavior hierarchy (Figure 1 are associated with which subteams (Figure 2). In our case, the `Defend` behavior (under `WinGame`) is associated with the Defenders subteam. The `Attack` and `Go-back-to-our-base` behaviors are associated with the Attackers and SecondAttackers subteams.

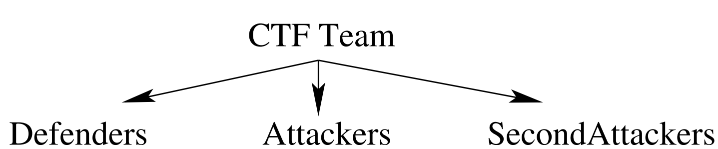

Fig. 2. Organization and Behavior Hierarchies for the CTF team.

The second arbitrator currently defined uses centralized preference-scheduling. Each agent sends its preferred selected behavior to a predefined team-leader. The team leader then greedily assigns behaviors to agents based on their preferences, and taking into account designer-specified constraints (expressed as the number of agents that must be assigned to each behavior) and then sends its decisions to the agents (which are assumed to accept them unconditionally). For instance, an agent may prefer to execute the Defend behavior if it is closer to the home base, or may prefer to execute the Get-Flag behavior if it has sufficient stamina. It will communicate its preference to the team leader, and will shortly thereafter be assigned a behavior (based on the team-leader's decisions) which it will then select for execution. In essence, this allows for dynamic formation and re-formations of teams by different agents, according to their preferences and the constraints imposed by the designer. All communications are of course automated by the arbitrator, which is triggered automatically when the agent comes to a decision point.

Unlike in previous teamwork models, the designer can mix and match arbitrators within the behavior hierarchy: When the agents have a selection to make, the designer-specified arbitrator is called to manage the actual arbitration procedure, and then the resulting joint decision is returned to each agent so that they begin execution of their agreed-upon behavior. In our experience, role-based arbitration is faster (since agents know their roles, the only communication requirements are those of synchronizing the team variables). However, preference-based arbitration tends to better take into account detailed situation features such as distances to goal locations, individual agent health, etc.

3.4 SCORE Behavior Language: Tying it all together

We defined a simple language in which team behaviors and arbitration can be specified. A single file contains all the behaviors of the team, specified as blocks of behavior data is read by each agent, and parsed by SCORE. It specifies the behavior names, its children and following behaviors, the selection and termination conditions, and arbitrators and associated constraints. As an example, a single behavior in the file is presented in Figure 3:

The fields and keywords used in Figure 3 are explained below:

startswhen *<condition>*. The *startswhen* keyword is followed by a condition which uses a Boolean expression containing variables in the shared team state. These specify selection conditions. When arbitrating over the behavior, the selection conditions are checked, in order to determine whether the behavior is selectable.

endswhen *<condition>*. The *endswhen* keyword is similar to the startswhen keyword, but controls the termination conditions, upon which a particular behavior should cease execution.

children *<child 1> <child 2> ... <child n>*. The *children* keyword describes the list of child behaviors reachable from the current behavior.

```
behavior Explore
  startswhen (var ourBaseKnown == NULL || var theirBaseKnown == NULL)
  endswhen (var ourBaseKnown != NULL && var theirBaseKnown != NULL)

  following WinGame
  children ExploreOurBase ExploreTheirBase

  arbitrator role
  constraint attacker
  constraint ExploreTheirBase
  constraint defender
  constraint ExploreOurBase
end behavior
```

Fig. 3. An example specification of a team behavior in the SCORE language.

following <*behavior name*>. The *following* keyword defines the next same-level behavior that follows the current behavior.

arbitrator <*arbitrator name*>. The *arbitrator* keyword defines the arbitrator that should be run at the current behavior, when deciding which of its alternative decompositions should be taken.

constraint <*string*>. The *constraints* are dependent on the arbitrator specified. There can be multiple constraint statements within the body of a single behavior, each is parsed as a string and passed as a vector of strings to the arbitrator. For role arbitration, the constraint field specifies which subteams are to select which child behavior. For preference-based arbitration, the constraints field specifies how many agents are required to select each behavior (to prevent a scenario where all agents pick one of the children, yet joint execution requires some to select the other children behavior as well).

4 Experiments

We evaluate SCORE and context-dependent arbitration policies in a set of experiments in which we compared the performance of CTF-playing teams using context-free and context-dependent arbitration policies. The experiments consisted of three sets of CTF games, each consisting of 5 games. Each set of 5 games used a different arbitration policy: a context-free role-based arbitration policy, a context-free preference-based arbitration policy, and a context-dependent arbitration policy mixing role-based and preference-based arbitrators. In all other respects, all of the game settings were identical: the same fixed opponent and simulated physical environment was used, and the behavior hierarchy was identical as well. The tables below compare the results of the three game sets using different performance measures.

Table 1 shows the score-difference results for each set of games (i.e., the number of flags captured by the evaluated team minus the number of flags captured by the fixed opponent). Score-difference is the ultimate task performance measure in this domain. In the table, each column corresponds to the three experimental settings. Each row but the last provides the score difference results in a game. The last row shows the average score difference in each set of games. The table clearly shows that context-dependent arbitration policy results in higher—better—score difference than either of the context-free arbitration policies.

Role Arbitration	Preference Arbitration	Mixed (Context-dependent) Arbitration
3	-3	3
-1	-3	1
-1	-3	1
0	-3	1
0	-1	1
0.2	-2.6	1.4

Table 1. Score difference in the three experiment sets.

Table 2 presents a second set of measurements from the same games. The structure of the table is identical to Table 1, but the entries correspond to the Average-Time-to-Agreement (ATA) [6]. The ATA measures average arbitration time, where each arbitration interval is taken from the time the first agent terminates one joint behavior, until all agents select a new joint behavior. Here the trade-offs between arbitration policies begin to emerge. Role-based arbitration (first column) clearly results in shorter arbitration intervals, while preference-based arbitration (second column) results in arbitration intervals more than five times longer on average. The context-dependent arbitration used requires just over twice the arbitration time as role-based arbitration, yet as we have seen it results in significantly better results (in terms of task performance).

Role Arbitration	Preference Arbitration	Mixed (Context-dependent) Arbitration
0.649	7.360	1.807
1.124	6.181	1.602
1.012	5.475	3.895
1.025	3.998	2.012
0.857	4.857	2.012
0.933	5.574	2.266

Table 2. Average Time to Agreement in the three experiment sets.

A final performance measure is presented in Table 3. Here, we measure how many times per second was the opponent flag *reached* by the agents (though not necessarily successfully captured and taken back to base). A larger value here indicates greater success at reaching the flag, but taking into account the score difference, we will see that such success does not necessarily translate into overall task performance (i.e., captured flags). Indeed, the results show that the context-dependent arbitration policy shows reduced success at reaching the flag, compared to role-based arbitration. However, when taking into account the results in Table 1 it becomes clear that context-dependent arbitration results in much more success than role-based arbitration in translating each incursion into a successful capture. In other words, agents utilizing context-dependent arbitration reach the flag slower and less often, but they are more successful at capturing the flag once it is reached.

Role Arbitration	Preference Arbitration	Mixed (Context-dependent) Arbitration
0.016	0.010	0.010
0.011	0.012	0.008
0.015	0.008	0.012
0.021	0.010	0.011
0.021	0.014	0.013
0.017	0.011	0.011

Table 3. Flags reached per second in the three experiment sets.

To summarize, the results demonstrate a context-dependent arbitration policy can result in significant performance benefits compared to context-free arbitration policies, both in terms of overall task performance (Table 1) and ability to translate opportunities into results (Table 3). The context-dependent arbitration policy we have evaluated managed to provide these benefits while keeping arbitration time significantly shorter than preference-based arbitration (Table 2).

5 Related Work

Previous teamwork models have not explored context-dependent arbitration policies. True to *Joint Intentions* theory [2] upon which they are partially based, both STEAM [13] and GRATE* [5] uses a static procedure where any agent is allowed to cause termination of a joint behavior, but a recognized team-leader selects the next joint behavior to be executed by all team-members. In STEAM, such a team-leader is given by designer-provided ranks. In GRATE*, the team-leader is the agent which originated the problem-solving activity for the team. COLLAGEN [11,9], a teamwork model specialized for intelligent user-interface tasks, allows both the user and its agent collaborator to propose tasks for joint execution, but naturally leaves conflict resolution to the user. ALLIANCE [10] uses an opportunistic agenda-based decision making. Every agent is free to select and de-select individual tasks, while communicating their decisions to their peers. This allows for great flexibility and fault-tolerance, ALLIANCE's explicit goals. On the other hand, models such as STEAM and ALLIANCE have mechanisms for failure detection and recovery, while SCORE currently lacks any such mechanisms.

Conceptually, work on the theory of negotiation by argumentation has recognized that different argumentation techniques (e.g., threat, promise, or appeal) are possible in different settings [8]. However, very little work has been done in terms of confirming such theoretical distinctions with empirical results, or of translating the theoretical distinctions into algorithms to be used in practice (though see [14] for promising empirical work). Our work does not provide a systematic exploration of argumentation techniques: Instead, it provides empirical evidence that even within relatively stable organizational settings (here, a team), a variety of appropriately used conflict resolution techniques (of which argumentation provides several special cases) is preferably to using a single method.

6 Summary and Future Work

We presented context-dependent arbitration policies in deploying multi-agent teams in computer games. To evaluate their usefulness, we presented SCORE, a state-of-the-art teamwork model whose key novelty is the ability to use different multi-agent decision-making procedures (arbitrators) depending on the context of task execution. We provided motivation for using SCORE in a complex, 3D environment where teams of agents are engaged in an adversarial game of Capture-the-Flag. We used this environment to empirically evaluate a context-dependent arbitration policy compared to context-free arbitration policies. The results show that context-dependent arbitration can significantly out-perform context-free arbitration.

We are currently exploring several future development directions. First, as each agent executes a different copy of the selected arbitrator, agents may get into failures where they differ in their selection of an arbitrator, or in the arbitration results. This opens up significant challenges which we hope to address in future work. In addition, we are planning to extend the SCORE behavior language and to fully develop a GUI development system for it, which allows users to drag-and-drop behaviors together to visually see the layout of the team program and the transitions that the agents will make. We also plan to offer a library of arbitration procedures and to examine systematically their empiric performance in the CTF domain.

References

1. T. Ando. Refinement of soccer agents' positions using reinforcement learning. In Hiroaki Kitano, editor, *RoboCup-97: Robot soccer world cup I*, volume 1395 of *LNAI*, pages 373–388. Springer-verlag, 1998.
2. Philip R. Cohen and Hector J. Levesque. Teamwork. *Nous*, 35, 1991.

3. R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence*, 1987.
4. Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group actions. *Artificial Intelligence*, 86:269–358, 1996.
5. Nicholas R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.
6. Gal A. Kaminka and Milind Tambe. Robust multi-agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research*, 12:105–147, 2000.
7. Gal A. Kaminka, Manuela M. Veloso, Steve Schaffer, Chris Sollitto, Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, , and Sheila Tejada. GameBots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45, January 2002.
8. Sarit Kraus, Sycara Katia, and Amir Evenchik. Reaching agreements through argumentation: a logical model and implementation. *Artificial Intelligence*, 104(1–2):1–69, 1998.
9. Neal Lesh, Charles Rich, and Candace L. Sidner. Using plan recognition in human-computer collaboration. In *Proceedings of the International Conference on User Modelling (UM-99)*, Banff, Canada, 1999.
10. Lynne E. Parker. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, April 1998.
11. Charles Rich and Candace L. Sidner. COLLAGEN: When agents collaborate with people. In W. Lewis Johnson, editor, *Proceedings of the International Conference on Autonomous Agents*, pages 284–291, Marina del Rey, CA, 1997. ACM Press.
12. Peter Stone, Manuela Veloso, and Patrick F. Riley. The CMUnited-98 champion simulator team. In *RoboCup-98: Robot soccer world cup II*, pages 61–76. Springer-verlag, 1999.
13. Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.
14. Milind Tambe and Hyuckchul Jung. The benefits of arguing in a team. *AI Magazine*, 20(4):85–92, 1999.
15. Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 2000.